NPS-CS-06-010

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

Design of Preliminary Experiments with the Sun Java Real-Time System

by

T. S. Cook, D. Drusinsky, J. B. Michael, T.W. Otani, and M. Shing

20 May 2006

Approved for public release; distribution is unlimited

Prepared for: Missile Defense Agency
7100 Defense Pentagon
Washington, D.C. 20301-7100

NAVAL POSTGRADUATE SCHOOL
Monterey, California 93943-5000

This report was prepared for the Missile Defense Agency and funded by the Missile Defense Agency.

This report was prepared by:

Thomas Otani
Associate Professor of Computer Science
Naval Postgraduate School

Reviewed by:	Released by:
Peter J. Denning, Chairman Department of Computer Science	Leonard A. Ferrari Associate Provost and Dean of Research

REPORT DOCUMENTATION PAGE		*Form Approved OMB No. 0704-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.		
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** May 20, 2006	**3. REPORT TYPE AND DATES COVERED** Technical Report
4. TITLE AND SUBTITLE: Title (Mix case letters) Design of Preliminary Experiments with the Sun Java Real-Time System		**5. FUNDING NUMBERS** MD6080101P1916
6. AUTHOR(S) Thomas S. Cook, Doron Drusinsky, James Bret Michael, Thomas W. Otani, and Man-Tak Shing		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000		**8. PERFORMING ORGANIZATION REPORT NUMBER** NPS-CS-06-010
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) Missile Defense Agency, 7100 Defense Pentagon, Washington, DC 20301-7100		**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this report are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.		
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution unlimited.		**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

There is an increasing interest in recent years to use the Java^{TM} programming language for implementing real-time systems. Recent advances in the Real-Time Specification for Java (RTSJ) have resulted in the introduction of new means for creating predictable real-time environments for Java programs. However, these new features also make the Java semantics more complex and the run-time behavior of the Java programs more difficult to analyze.

In this technical report, we describe a number of preliminary experiments we performed to study the features of the Sun Java Real-Time System (RTJ 1.0). We designed these experiments to verify the viability of the Real-Time Java language for the implementation of the Global Integrated Fire Control System (GIFC)—a component of the C2BMC element of the Ballistic Missile Defense System (BMDS).

Our preliminary experiment shows that it is preferable to use only the Real-Time Java threads that use the heap memory and not the no-heap real-time threads for the GIFC software. However, such architecture cannot be implemented by using RTJ 1.0. Further experiments are needed to determine if the preferred architecture can be implemented with the upcoming RTJ 2.0, which will give programmers more control over the priority of the garbage collection.

14. SUBJECT TERMS Real-time system, Java programming language, Garbage collection, Ballistic Missile Defense System, Global Integrated Fire Control, Advanced Battle Manager			**15. NUMBER OF PAGES** 22
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Design of Preliminary Experiments with Sun Java Real-Time System

T. S. Cook, D. Drusinsky, J. B. Michael, T. W. Otani, and M. Shing

Abstract

There is an increasing interest in recent years to use the Java^{TM} programming language for implementing real-time systems. Recent advances in the Real-Time Specification for Java (RTSJ) have resulted in the introduction of new means for creating predictable real-time environments for Java programs. However, these new features also make the Java semantics more complex and the run-time behavior of the Java programs more difficult to analyze.

In this technical report, we describe a number of preliminary experiments we performed to study the features of the Sun Java Real-Time System (RTJ 1.0). We designed these experiments to verify the viability of the Real-Time Java language for the implementation of the Global Integrated Fire Control System (GIFC)—a component of the C2BMC element of the Ballistic Missile Defense System (BMDS).

Our preliminary experiment shows that it is preferable to use only the Real-Time Java threads that use the heap memory and not the no-heap real-time threads for the GIFC software. However, such architecture cannot be implemented by using RTJ 1.0. Further experiments are needed to determine if the preferred architecture can be implemented with the upcoming RTJ 2.0, which will give programmers more control over the priority of the garbage collection.

1.0 Overview

The BMDS battle-management (BM) software is a real-time set of system functionality that addresses warfighter usage. Key characteristics of the BM will include the following: (1) a globally-distributed network, (2) an operational battlespace that includes land, sea, air, and space, (3) capability to address multiple targets that can threaten a specific theater of operations or region of the world, (4) management of concurrent battlespace activities, (5) some level of automated decision making regarding the release or hold of lethal weapons, and (6) stringent requirements for high levels of trustworthiness of the systems that provide BMD capabilities due to the fact that the threats to be encountered consist of weapons of mass destruction (WMD). Item number six makes unpredictable system behavior untenable from the public-policy, functional, and safety perspectives.

This is a progress report on our research to support the Missile Defense Agency (MDA) in developing and applying advanced technology in support of developing the Global Integrated Fire Control System (GIFC)—a component of the C2BMC element of the Ballistic Missile Defense System (BMDS). Our research is driven by the needs of the Missile Defense Agency to prepare for the delivery of the GIFC to PACOM for use in the "Terminal Fury" Exercise, which will take place in summer 2007. The exercise will be used to simulate a large-fight threat space with coordinated attacks by adversaries against the United States, its allies and friends. The GIFC and the rest of the C2BMC components must be able to successfully execute the kill chain (i.e., detection through assessment of kill) for each of the high-priority threat objects (to include cruise missiles, ballistic missiles, and air threats) tracked by the BMDS sensor networks.

Here we describe our initial experiments to study the viability of our preliminary software architecture design for the real-time GIFC.

2.0 RTJ v1.0 and v2.0

We began using RTJ 1.0 (Sun reference implementation called Mackinac) in our study. The defining feature of RTJ 1.0 that severely affects the implementation of MDS is the independence of the system's garbage collector against other real-time threads. With RTJ 1.0, the priority we assign to a real-time thread does not affect its scheduling relative to the garbage collector. In other words, even if we assign the highest possible priority to a real-time thread, it can get interrupted by the garbage collector. We ran an experiment to verify this system behavior (Experiment No. 1). Since the garbage collector cannot be controlled programmatically, the only recourse we have with RTJ 1.0 is to run the deadline-sensitive stateless discriminator as a no-heap real-time thread. Because it does not use any heap memory, it will never be interrupted by the garbage collector. We describe in the next section the experiment (Experiment No. 2) that uses no-heap real-time threads.

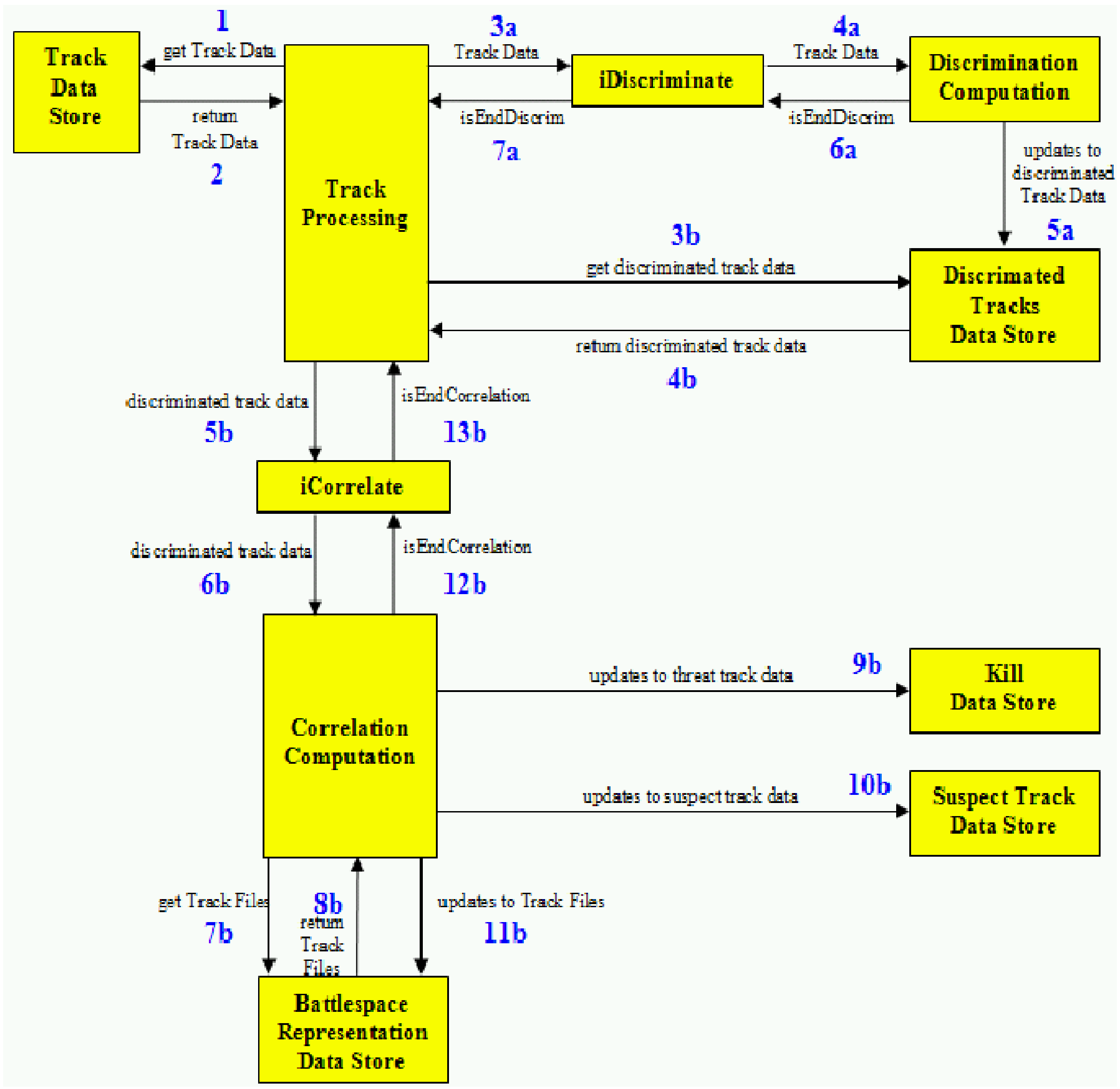
We visited Sun Microsystems in early March 2006 to consult with the leaders of the RTJ development team. We learned about the enhancement to RTJ 2.0 that allows the programmatic control of the garbage collector. RTJ 2.0 permits programmers to assign the scheduling priority of real-time threads relative to the priority of the garbage collector. For a time-critical real-time thread, we can assign the priority higher than the one for the garbage collector so this real-time thread does not get interrupted by the garbage collector.

3.0 ABM Track Processing

One of the primary components of the GIFC is the Advanced Battle Manager (ABM). The ABM is a real-time, reactive system. The ABM component systems continuously interact with their environment under tight timing constraints. Both the inputs and outputs of these component systems must satisfy timing constraints imposed by the BMDS. For the purposes of experimentation, we chose to try out different strategies for designing real-time functions in RTJ by developing software for the tracking function of the ABM, as depicted in Figure 1. The primary functions of the ABM tracker are as follows:

- Interface with ABM and non-organic sensors
- Discriminate own sensor data
- Correlate sensor data
- Generate fused tracks

FIGURE 1. Notional model of tracking function of the ABM (from D. S. Caffall, Developing Dependable Software for a System-Of-Systems, Ph.D. thesis, Naval Postgraduate School, Monterey, Calif., Mar. 2005)


Track Data Store
1 get Track Data
return Track Data 2
Track Processing
3a Track Data
iDiscriminate
4a Track Data
Discrimination Computation
isEndDiscrim 7a
isEndDiscrim 6a
updates to discriminated Track Data 5a
3b get discriminated track data
Discrimated Tracks Data Store
return discriminated track data 4b
discriminated track data 5b
isEndCorrelation 13b
iCorrelate
discriminated track data 6b
isEndCorrelation 12b
Correlation Computation
updates to threat track data 9b
Kill Data Store
updates to suspect track data 10b
Suspect Track Data Store
get Track Files 7b
8b return Track Files
updates to Track Files 11b
Battlespace Representation Data Store

4.0 Experiments

One of the challenges we face our study is the scarcity of available references. Our main sources of information about RTJ are Bollela [BOLL], Dibble [DIBB], and Wellings [WELL]. Because of the limited references to cross-check our findings, we decided to verify every key piece of information given in said references.

In this section, we present the main experiments we performed. With these experiments, we tested the architecture of our basic design ideas for the real-time GIFC. We have executed numerous other test programs, but they are mainly for the purpose of understanding the RTJ system and will not be included in the discussion here. Also, we will not discuss the minor tests and different variations of the three experiments presented here.

All of the experiments described in this report were run under RTJ 1.0. We will rerun tehse and additional experiments under the alpha release of RTJ 2.0 and report the results in a followup technical report.

4.1 Experiment No. 1: Testing the Effect of Garbage Collection

We ran a small test program to verify that the garbage collector will interrupt even the highest priority real-time thread. The main class **RTComputation_LinkedListAllocation**, a grandchild of **javax.realtime.RealtimeThread**, creates 20 instances of itself. Each instance will allocate an array of **BigInteger** objects and add this array to a linked list. The run method of this thread repeats this process for N (= 20 for the sample execution) times. This simulates the thread doing some work.

We run the program with the option **-verbose:gc** so we can see the garbage collection activity. The following is a sample output from the program:

```
Free Memory: 3491152
Elapsed:    (6 ms, 960583 ns)

Free Memory: 3069320
Elapsed:    (2 ms, 120833 ns)


Free Memory: 2654576
Elapsed:    (2 ms, 320250 ns)


Free Memory: 2239832
```

```
Elapsed:    (2 ms, 75667 ns)


Free Memory: 1825344
[GC 2046K->349K(3520K), 0.0112029 secs]
Elapsed:    (18 ms, 478249 ns)


Free Memory: 3147856
Elapsed:    (4 ms, 671666 ns)


Free Memory: 2733112
Elapsed:    (1 ms, 283833 ns)


Free Memory: 2318368
Elapsed:    (1 ms, 257166 ns)


Free Memory: 1903624
Elapsed:    (1 ms, 339583 ns)


Free Memory: 1488880
[GC 2396K->775K(3520K), 0.0049108 secs]
Elapsed:    (8 ms, 136417 ns)


Free Memory: 2734680
Elapsed:    (0 ms, 846333 ns)


Free Memory: 2319936
Elapsed:    (1 ms, 329666 ns)


Free Memory: 1905192
Elapsed:    (0 ms, 968584 ns)


Free Memory: 1490448
Elapsed:    (0 ms, 978333 ns)


Free Memory: 1075704
[GC 2822K->1201K(3520K), 0.0048332 secs]
Elapsed:    (6 ms, 423333 ns)


Free Memory: 2321384
Elapsed:    (0 ms, 912250 ns)
```

```
    Free Memory: 1906640
    Elapsed:    (0 ms, 916750 ns)


    Free Memory: 1491896
    Elapsed:    (0 ms, 954083 ns)


    Free Memory: 1077152
    Elapsed:    (0 ms, 968167 ns)


    Free Memory: 662408
    [GC 3248K->1626K(3776K), 0.0054497 secs]
    [Full GC 1626K->424K(3776K), 0.0077447 secs]
    Elapsed:    (15 ms, 164833 ns)
```

The output lines

```
Free Memory: 3491152
Elapsed:      (6 ms, 960583 ns)
```

indicate the amount of free memory in bytes and the elapsed time of running one thread to completion. In this sample program, we create 20 such threads. The output lines

```
[GC 3248K->1626K(3776K), 0.0054497 secs]
[Full GC 1626K->424K(3776K), 0.0077447 secs]
```

indicate the garbage collection activity. The label **GC** indicates normal garbage collection and **Full GC** indicates a more complete garbage collection. The legend for the output line is as follows:

```
[GC xK -> yK (zK), t secs]

xK - size of live objects before GC
yK - size of live objects after GC
zK - total space available
t - time taken to complete the GC
```

This experiment confirms that we do not have programmatic control of the garbage collector. The system will run it “whenever” it deems necessary regardless of the priority of the running real-time thread. As shown in the sample output, there is a huge disparity in the elapsed time, ranging from the minimum of (0 ms, 846333 ns) to (18 ms, 478249 ns). We conclude from this result that we have no option but to run the deadline-sensitive task as a no-heap real-time thread.

4.2 Experiment No. 2 : Running No-Heap Real-time Thread (NHRTT)

Since the regular Real-time Thread (RTT) gets interrupted by the garbage collector, with RTJ 1.0, we must run any deadline-sensitive thread as a no-heap real-time thread (NHRTT). In this experiment, we verify the correct procedure for creating NHRTTs and that NHRTT does not get interrupted by garbage collection. Creating no-heap real-time threads correctly is one of the critical aspect when dealing with NHRTTs. It is not just a matter of calling the new operation for NHRTT.

The standard technique for creating a NHRTT is to let an object (thread) that creates the NHRTT enter the ImmortalMemory area. In this experiment, we define a Runnable object named NhCreator. The sole purpose of this object is to create a NHRTT and run it. A NhCreator itself is created in a heap, but we make it “enter” into an ImmortalMemory:

```
ImmortalMemory.instance().enter(new NhCreator())
```

Once it enters an ImmortalMemory, any object (thread) it creates will be allocated in the immortal memory (or the scoped memory, which can be specified at the time a NHRTT is created).

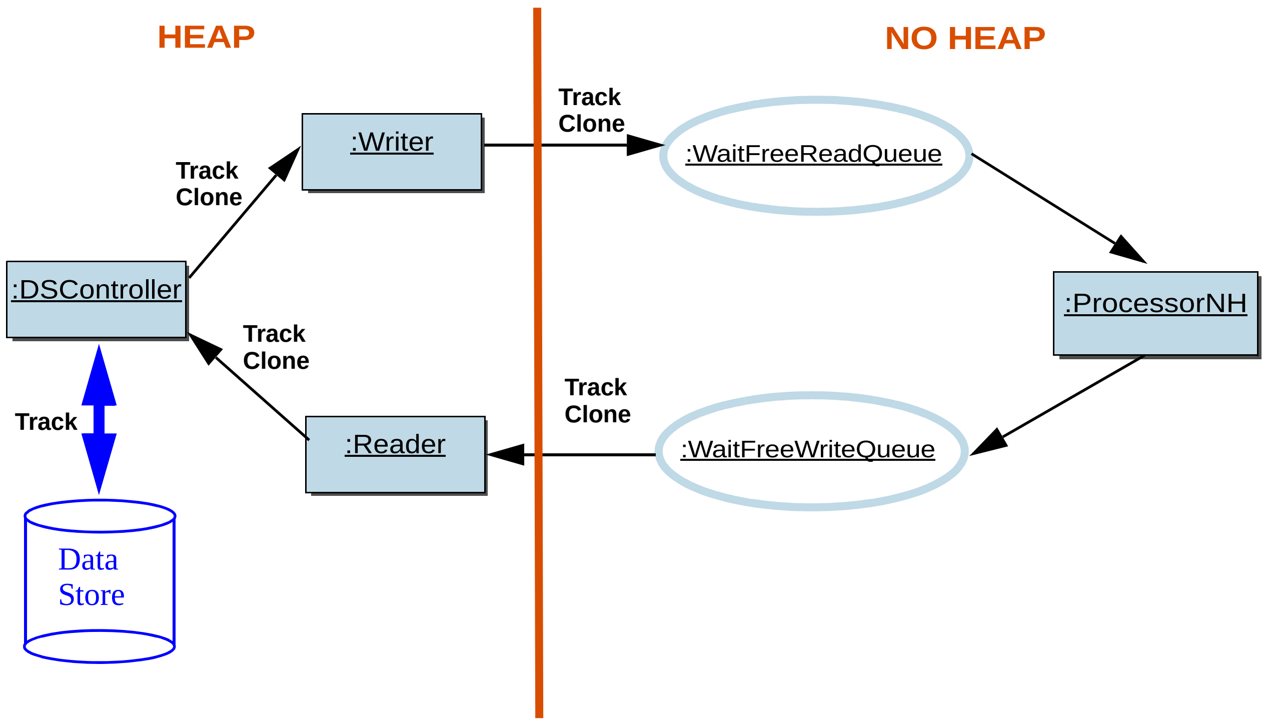
4.3 Experiment No 3: Testing Our Heap/No-Heap Combo Design

In this experiment, we explore the viability of one of the two main design options we consider for the MDS. To avoid the untimely interruption by the garbage collector, we propose to execute the track discriminator as a NHRTT. The data store for the tracks and the object (RTT) that manages this data store are in the heap memory. The track discriminators are NHRTTs, and they reside in an ImmortalMemory. The tricky aspect of this Heap/No-heap architecture is the communication link setup between the two types of objects (those in Heap and those in ImmortalMemory). The track objects are in heap, but we must pass this object to the no-heap track discriminators in the ImmortalMemory. No-heap threads, of course, cannot access any object in heap (if such thing is allowed, no-heap threads would be impacted by the garbage collector). Thus, we must set up the communication link between the two by using WaitFreeReadQueue and WaitFreeWriteQueue. We wrote a program to test the architecture shown in Figure 2:

4.3.1 ProcessorNH

ProcessorNH does not have to wait to get (read) data from the WaitFreeReadQueue and does not have to wait to put (write) data to the WaitFreeWriteQueue. For each Track clone that comes out of the wait-free read queue, ProcessorNH creates a Discriminator to discriminate the

FIGURE 2. This diagram illustrates the use of the WaitFreeReadQueue and WaitFreeWriteQueue classes



track. The Discriminator will return the Track clone with its discrimination to the wait-free write queue.

4.3.2 DSController

DSController manages the Track data store. Since no-heap RTT cannot directly access objects in the heap memory, DSController creates and passes a clone of the Track object to ProcessorNH. The actual communication is handled by the Writer. When a Track clone comes back from the ProcessorNH, via the Reader, DSController updates the corresponding Track object in the data store.

4.3.3 Reader and Writer

Writer receives a Track clone from DSController and passes it to the wait-free read queue. Writer can be blocked and wait until it can write the Track clone to the queue. The term “wait-free” is relative to the read of this queue, that the reader of this queue does not wait. Reader continually monitors the wait-free write queue for any result. Reader will also fetch the next available Track clone from the queue and pass the result back to the DSController so it can update the data store.

4.4 Experiment No. 4: Testing Our All-Heap Design

Working with NHRTT is not easy. There are many pitfalls and hurdles software engineers and programmers must jump. With the upcoming RTJ 2.0, we should be able to run all objects (threads) in a heap because programmers will have a control over the garbage collection. In this All-Heap Design, instead of running the Discriminator as NHRTT, we will run it as a regular RTT, but assign a scheduling priority higher than the one assigned to the garbage collector. The key innovation of this design is the use of nominal result. The proposed architecture is in Figure 3 and the sequence diagram in Figure 4:

FIGURE 3. This class relationship diagram shows the relative priority of the four key classes in the proposed all-heap design. DiscriminatorNominal and DiscriminatorDeadlineHandler objects have a priority higher than and DSController and DiscriminatorStateless objects a priority lower than the priority of the garbage collector.

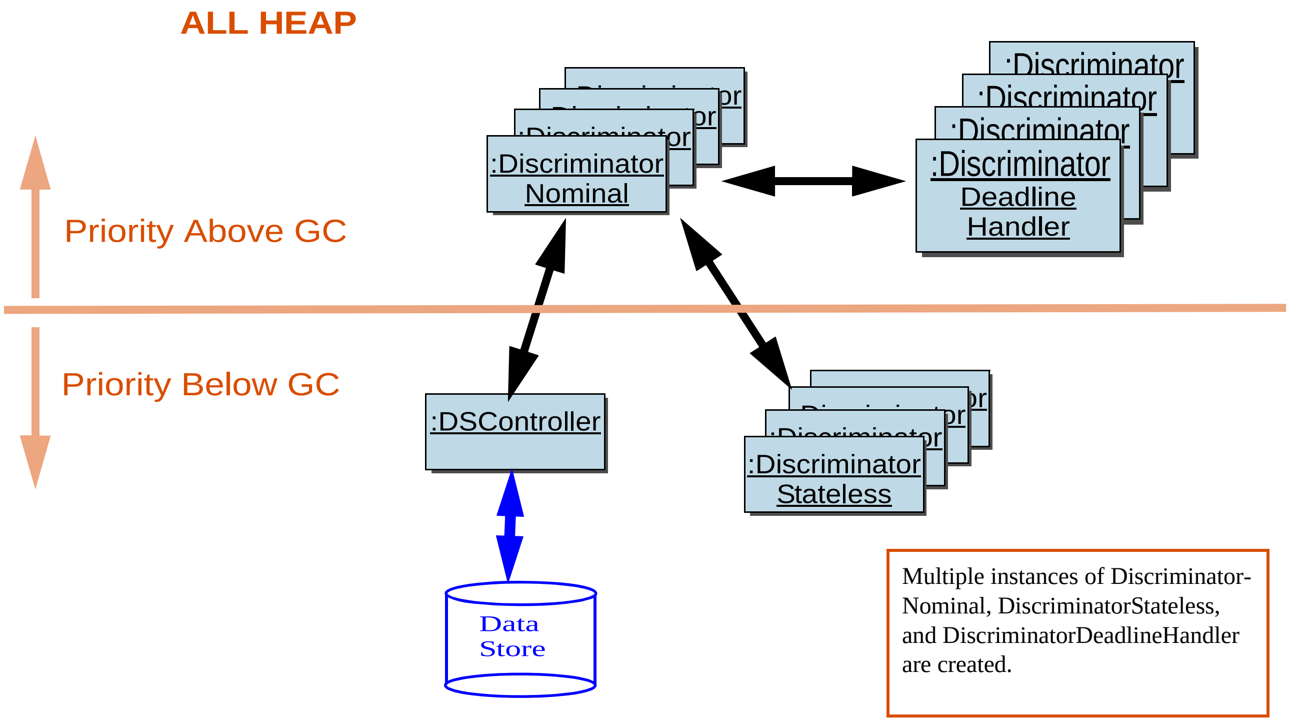
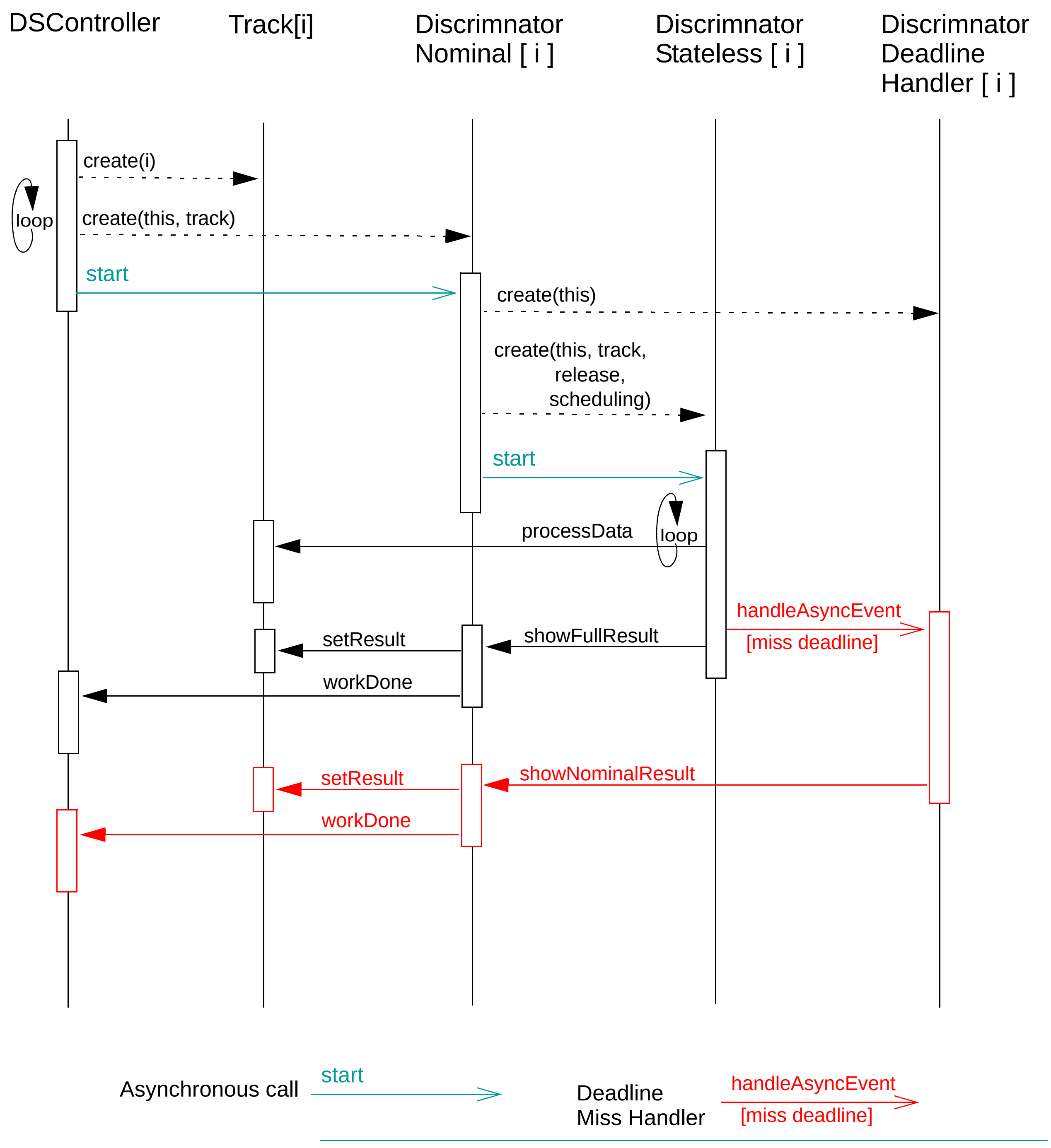

ALL HEAP
Priority Above GC
Priority Below GC
:Discriminator Nominal
:Discriminator Deadline Handler
:DSController
:Discriminator Stateless
Data Store
Multiple instances of Discriminator-Nominal, DiscriminatorStateless, and DiscriminatorDeadlineHandler are created.

FIGURE 4. This is the sequence diagram of the classes in the All-Heap design.



4.4.1 DSController

The main controller of the program. It creates N tracks, and for each track created, an instance of DiscriminatorNominal is assigned to it for discrimination.

4.4.2 DiscriminatorNominal

A DiscriminatorNominal object performs the discrimination operation on the given track. The actual work of discrimination is done by DiscriminatorStateless. The deadline is set and DiscriminatorDeadlineHandler is designated as its deadline miss handler.

4.4.3 DiscriminatorStateless

An instance of this class does the actual work of discrimination. When the full discrimination is completed, it calls its controlling DiscriminatorNominal to report the result.

4.4.4 DiscriminatorDeadlineHandler

When the set deadline is missed by the DiscriminatorStateless, it calls its controlling DiscriminatorNominal to report that the nominal result must be used.

4.4.5 Thread Priorities

DiscriminatorNominal’s priority is set to P, which is higher than the priority of GC. DiscriminatorStateless’s priority is set to Q, which is lower than the priority of GC. Priority of deadline miss handler DiscriminatorDeadlineHandler is set to P+c, where c >=1. A DiscriminatorDeadlineHandler object must have a priority higher than the one assigned to the thread it is interrupting.

NOTE: We do not have RTJ 2.0 yet. We only tested this architecture as much as possible under RTJ 1.0. We will develop further and perform detailed testing with RTJ 2.0 when we acquire it.

5.0 Multiprocessor Implementation of RTJ 2.0

During our meeting with SUN’s RTJ project members, we raised the question on the clock precision of the RTJ 1.0. They informed us that, although the Solaris 9 Operating System is non-real-time, RTJ 1.0 system bypasses the soft clock of the Solaris 9 Operating System and access the hardware clock directly. In doing so, the RTJ 1.0 real-time thread is able to operate accurately in the micro-second range.

In order to test the scalability of the proposed All-Heap Design, as outlined in Section 3.4, we will need a good estimate on the average execution time of the stateless algorithms that will be used by the MDS.

For example, assume that the track processing module control loop runs in a 2-second cycle and the stateless algorithm has an average execution time of 100 ms per track. A RTJ system running on a single processor can process at most 2000/100 = 20 tracks per cycle. On the other hand, if we have a more efficient stateless algorithm with an average execution time of, say, 10 ms per track, a RTJ system running on a single processor may be able to process up to 2000/10 = 200 tracks per cycle.

Since we expect that the track processing module has to process far more than 200 tracks per 2-second cycle, it is likely that the multi-processor implementation of RTJ 2.0 is required for the MDS. We will study this issue further.

6.0 Virtual Machine Internal Error

Throughout our experiments, we have encountered occasional virtual machine internal errors (Hotspot Virtual Machine Error) that complain about problematic threads. At this point, we do not know the source of the problem. It is possible that some coding error on our part is causing this erratic behavior. However, we believe they are truly the internal errors that should not occur because they occur sporadically and intermittently in different programs. We will monitor this internal error closely when we start using RTJ 2.0.

7.0 References

[BOLL] Bollella, Greg., et. al., *The Real-Time Specification for Java*, Addison-Wesley, 2000.

[DIBB] Dibble, Peter C., *The Real-Time Java Platform Programming*, Prentice-Hall, 2002.

[WELL] Wells, Andy, *Concurrent and Real-Time Programming in Java*, John Wiley & Sons, 2004.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   8725 John J. Kingman Rd., STE 0944
   Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library, Code 52
   Naval Postgraduate School
   Monterey, CA 93943-5100

3. Research Office, Code 09
   Naval Postgraduate School
   Monterey, CA 93943-5000

4. Dr. Butch Caffall
   Missile Defense Agency
   Washington, DC

5. LTC Jason Stine
   Missile Defense Agency
   Washington, DC

6. LTC Thomas Cook
   Naval Postgraduate School
   Monterey, CA

7. Dr. Doron Drusinsky
   Naval Postgraduate School
   Monterey, CA

8. Dr. Bret Michael
   Naval Postgraduate School
   Monterey, CA

9. Dr. Thomas Otani
   Naval Postgraduate School
   Monterey, CA

10. Dr. Man-Tak Shing
    Naval Postgraduate School
    Monterey, CA

11. Mr. Scott Pringle
Missile Defense National Team
Crystal City, VA

12. Mr. Erik Stein
Missile Defense National Team
Crystal City, VA

13. Mr. Tim Trapp
Missile Defense National Team
Crystal City, VA

14. Ms. Deborah Stiltner
Missile Defense National Team
Crystal City, VA

15. Mr. Dion Hinchcliffe
Missile Defense National Team
Crystal City, VA